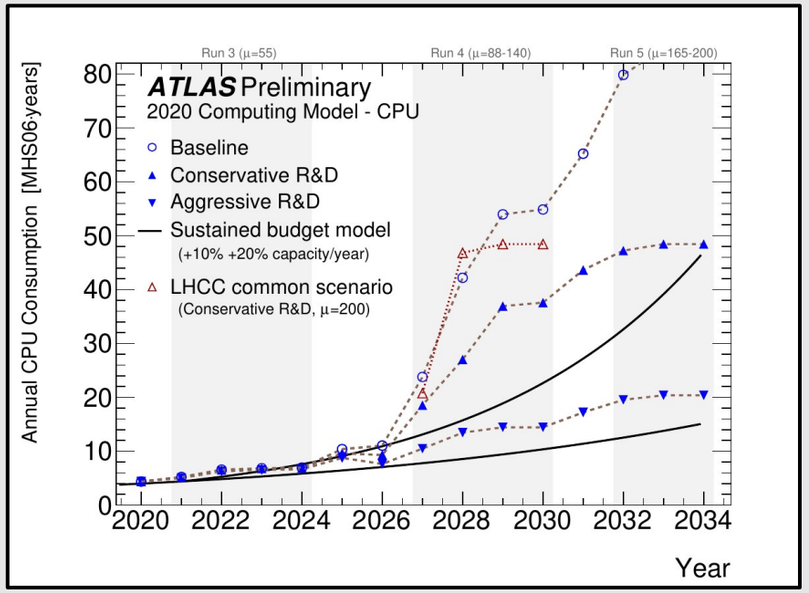
FastCaloSim Status

Charles Leggett

HEP-CCE/PPS All Hands
November 5 2020

Quick Motivation: Why FastCaloSim?

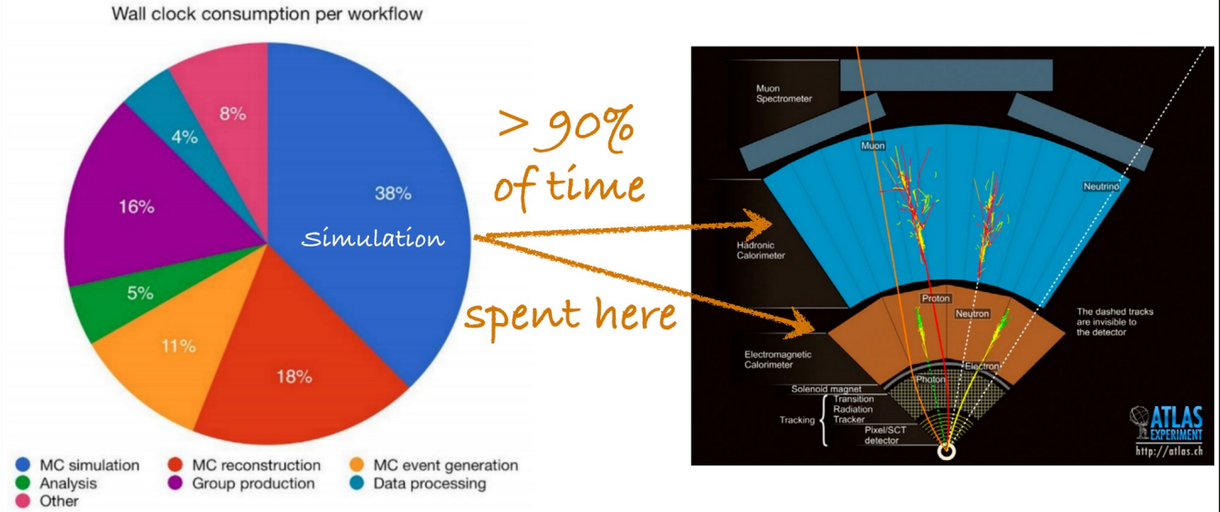
- ATLAS needs lots of simulation
  - Simulation is paramount for SM and background modeling in most analyses, as well as general detector and upgrade studies
  - A significant issue in Run-2 was the lack of MC-based statistics, and will only worsen in Run-3 and beyond without faster production



- A very large fraction of the simulation's computational budget is spent by the LAr Calorimeter
  - Parametrized simulation can speed things up enormously: FastCaloSim

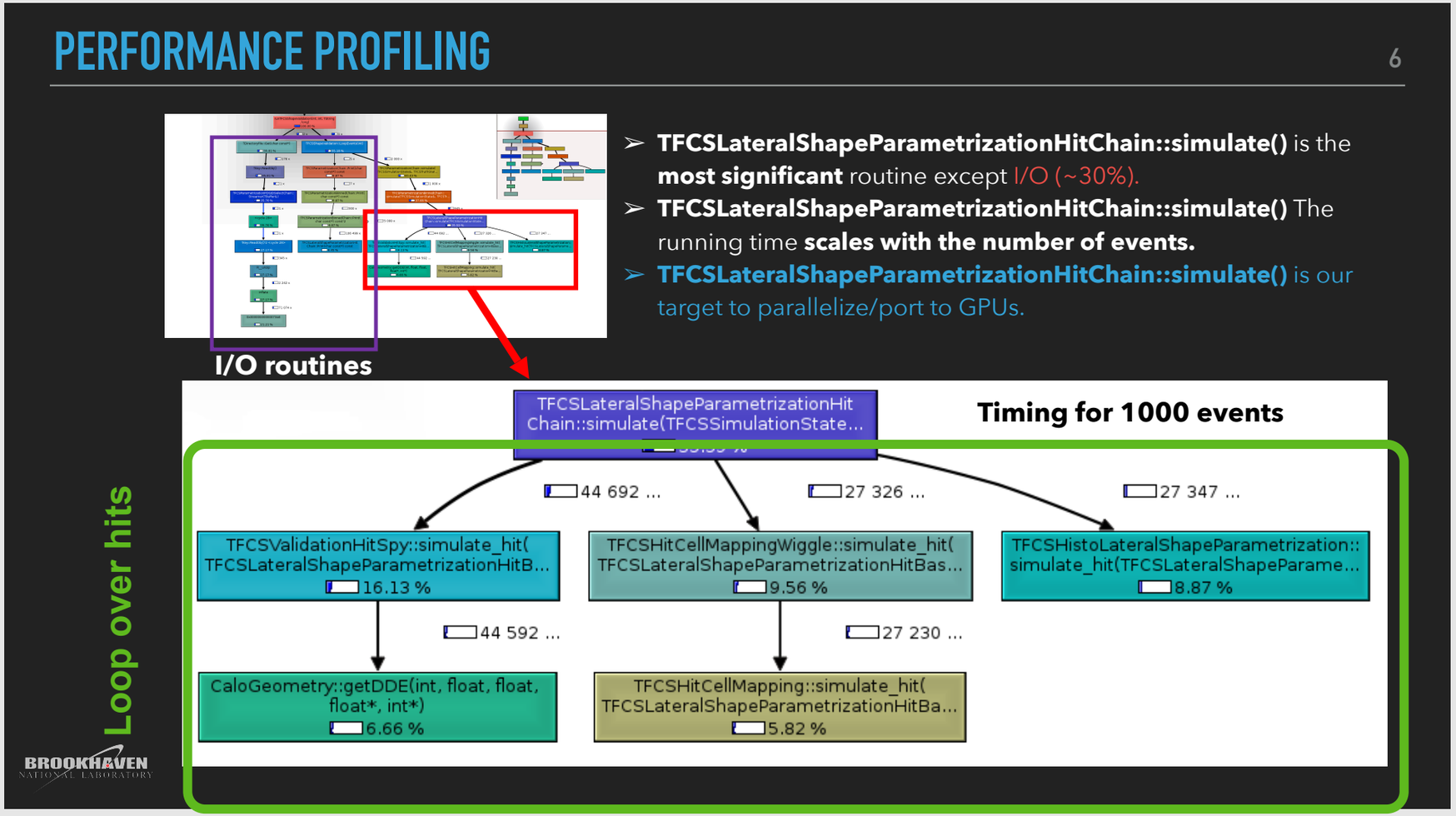
- FastCaloSim is small, self contained, few dependencies, already had a CUDA port

Calorimeter-dominated



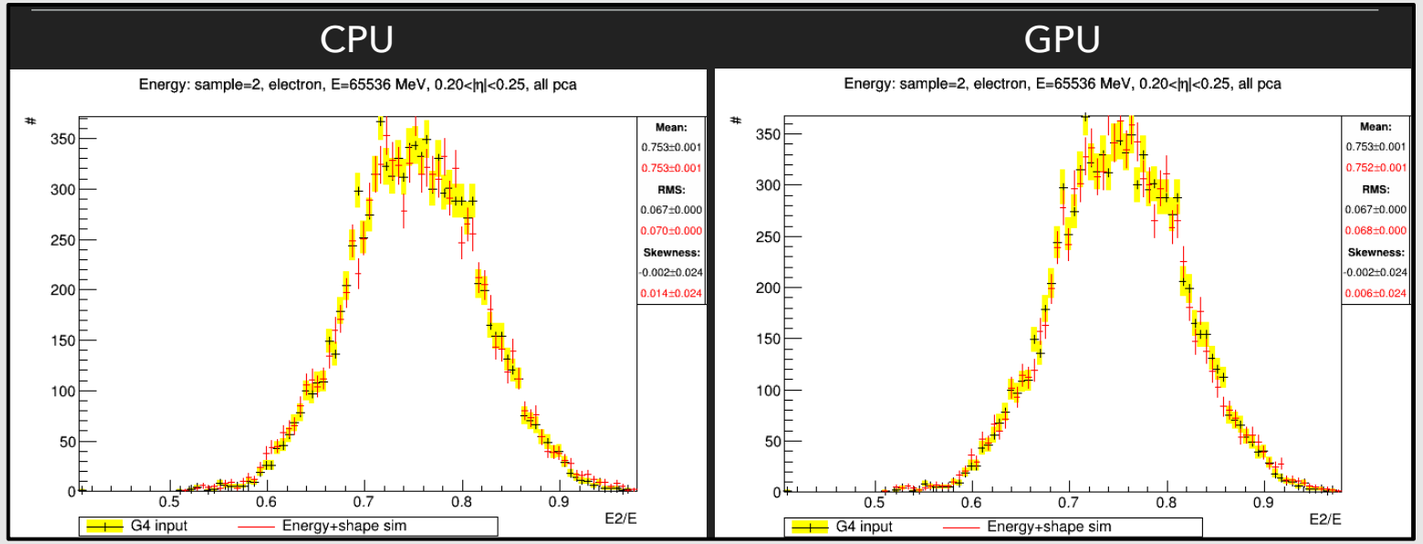
CPU Code Profiling

- LAr Calorimeter has massive inherent parallelism – lots of independent cells and associated tasks.
- Profiling studies identified likely hotspots that are paralellizable
- CUDA kernels created to run these parts on the GPU
  - modified data structures
  - reimplement Geometry and parametrization tables for GPU – no STL allowed
  - 3 kernels:
    - reinit memory
    - main simulation
    - reduction



GPU Validation

- CUDA has a very good random number generator (cuRAND)
  - FCS needs *lots* of random numbers
    - 3 per hit x ~5k hits per event
  - much faster than generating on CPU
  - but can't do bitwise comparisons with CPU – only statistical
  - after looking at lots of histograms, results look statistically equivalent

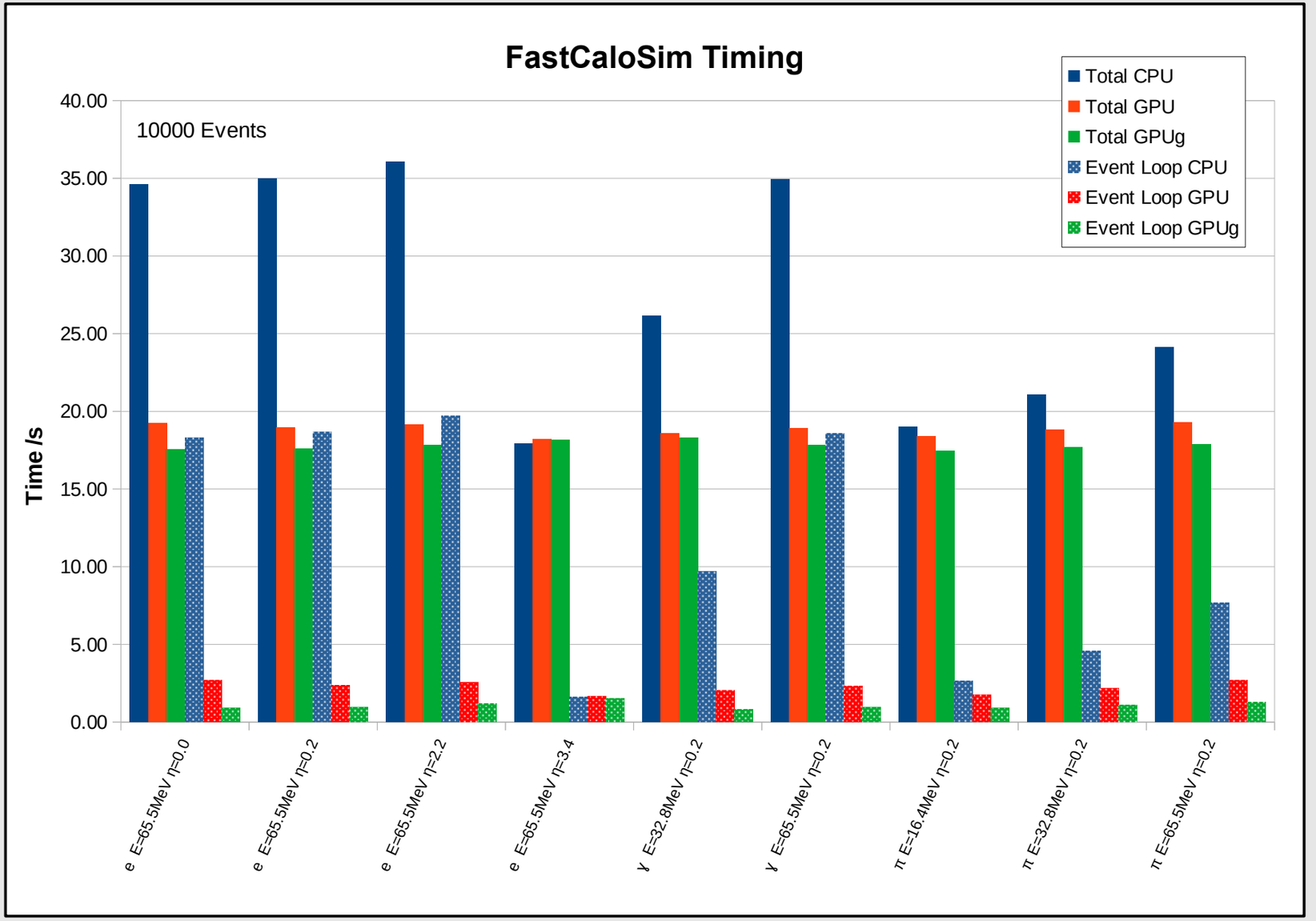


- If we sacrifice speed, we can generate random number on CPU, and transfer them to GPU, using these for all calculations on GPU
  - compared the results of 62 million hits in the Electron 64 GeV run
  - found only 2 hits calculations that ended up in different calorimeter cell
    - slightly different float rounding policies on CPU/GPU
  - if we use double precision variables for certain calculations, difference vanishes

- Confident that GPU code does the same thing as CPU

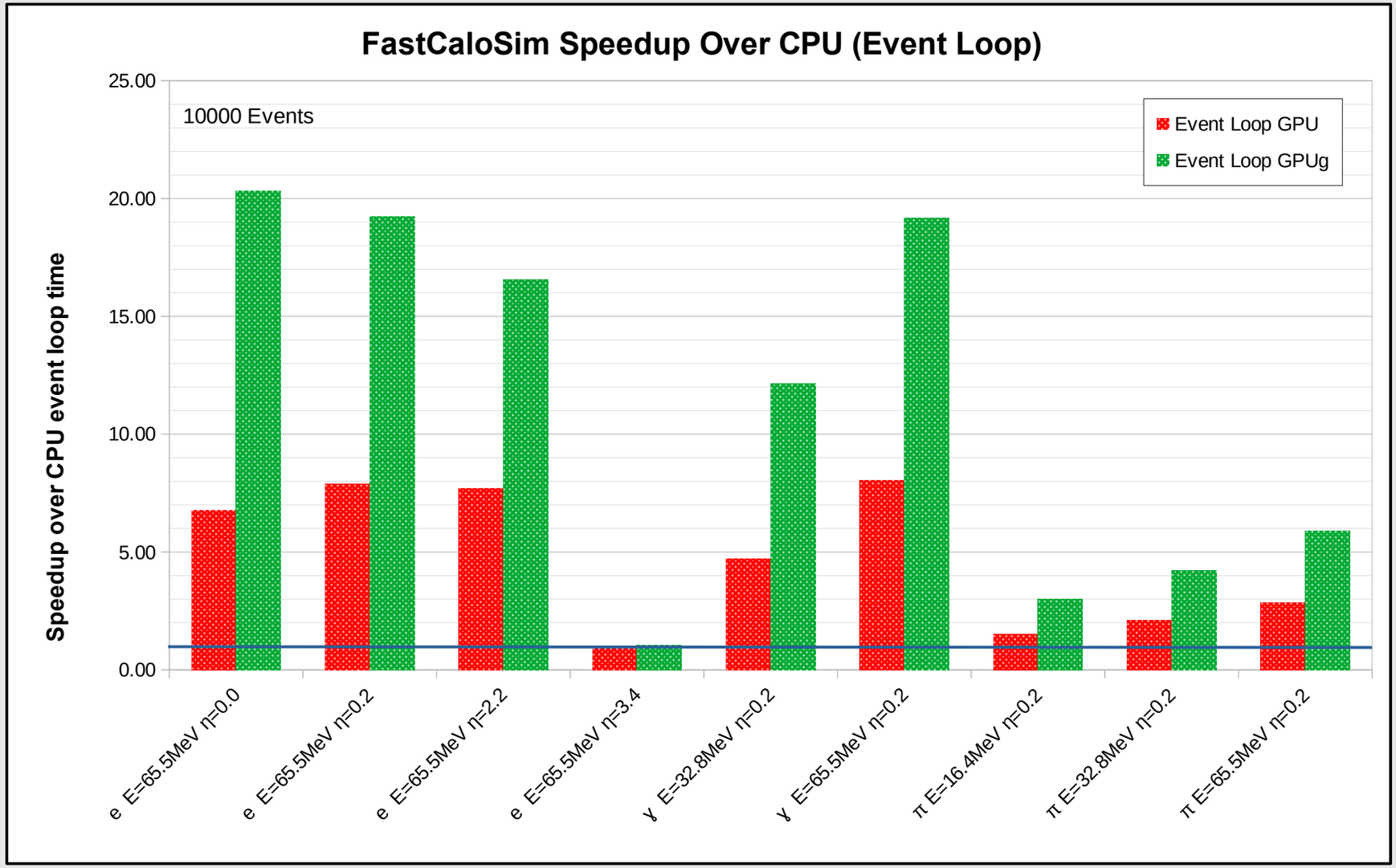
CUDA Performance Studies

- I/O to read/unpack parametrization files is expensive: ~15s of 30s
- Execution only offloaded if >500 hits, otherwise CPU is faster
- GPU kernels very short
  - launch latency limited
- Better performance if group work between multiple events to give more work to GPU



CUDA Performance Studies

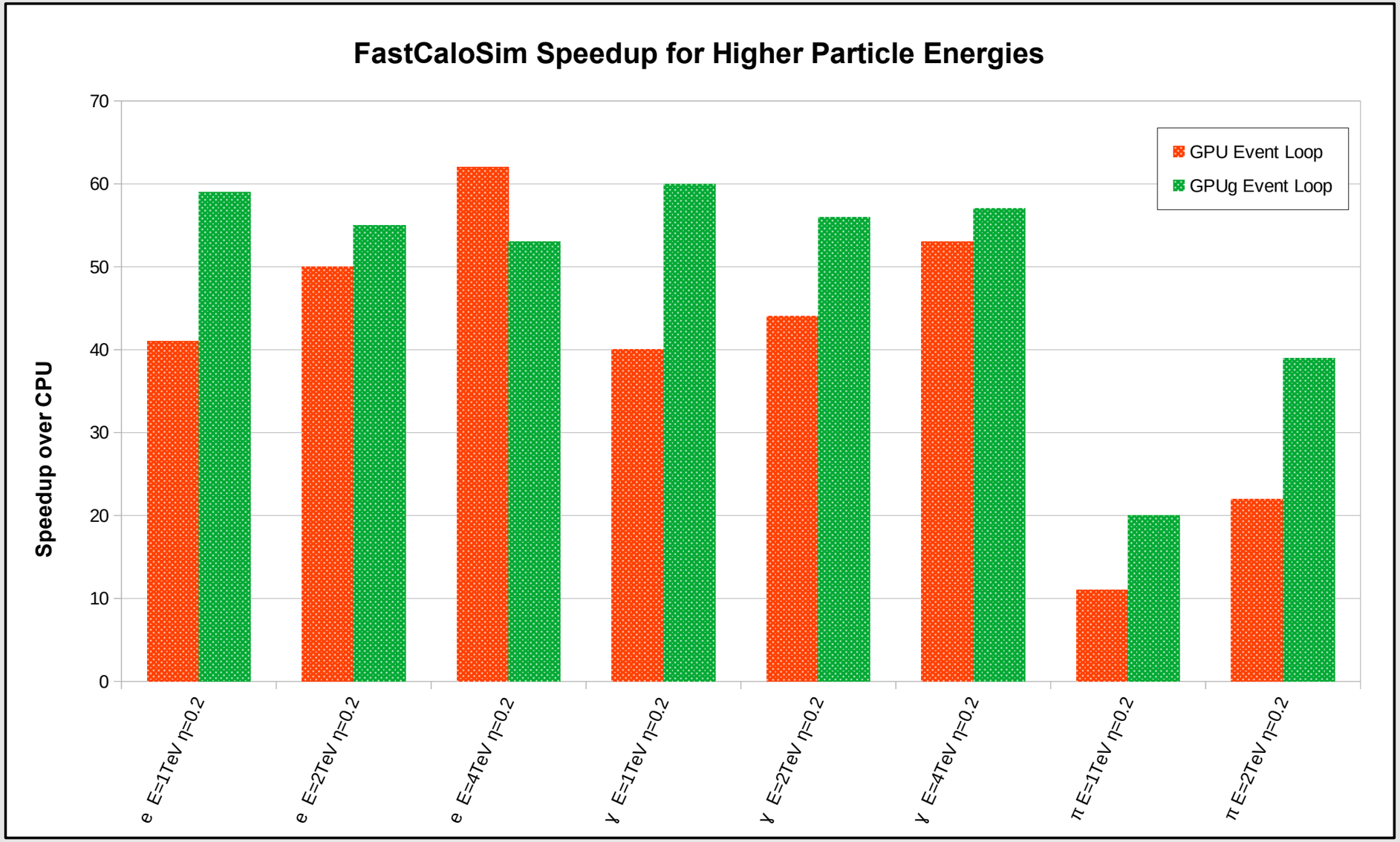
- I/O to read/unpack parametrization files is expensive: ~15s of 30s
- Execution only offloaded if >500 hits, otherwise CPU is faster
- GPU kernels very short
  - launch latency limited
- Better performance if group work between multiple events to give more work to GPU



CUDA Performance Studies

- GPU performs better for higher energy particles (more hits/work)

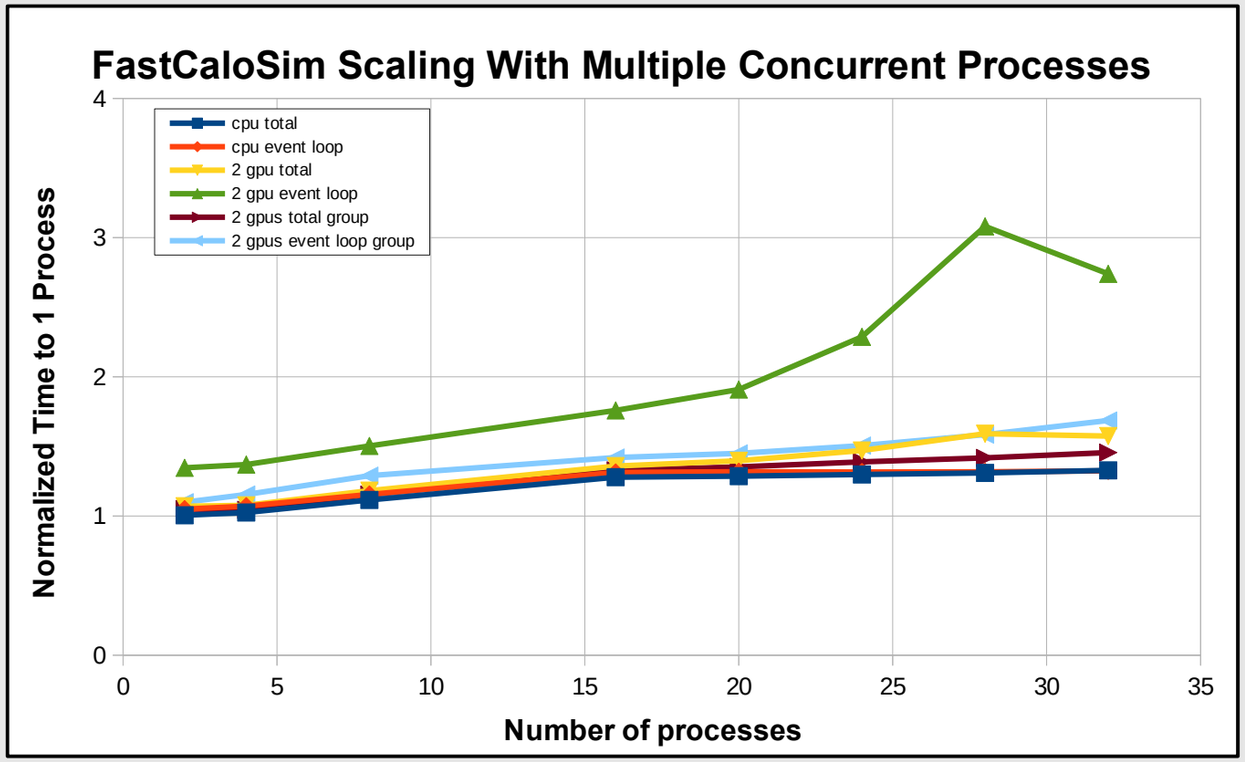
- Grouped work not as effective since regular GPU is already performant
  - need to send extra information to GPU when work is grouped between events



GPU Usage With CUDA

- In general, GPU resources are not well used
  - kernels are very short, dominated by launch latency overheads
  - work size is small, under-utilizing available GPU cores

- Can run multiple concurrent process all sharing one (or more) GPUs
  - use nvidia-cuda-mps-server to share 2 P100s between up to 32 processes
  - curve is mostly flat – nowhere near maxing out GPU resources
  - we can do the same with a V100 w/ 48GB and run 62 processes with little impact on performance



Porting to Kokkos

- Build infrastructure
  - Kokkos has decent CMake integration
  - requires separate binaries for each device backend (CUDA, HIP, Intel) or host parallel (pThread, OpenMP)
  - In theory you can run both device/host parallel backends in same code, but then you can't use the default execution space for your kernels: have to say which go where
- Shared libraries not compatible with device symbol relocation
  - if you want shared libs, all symbols in a kernel must be visible to one compilation unit
    - wrap kernels in one file that does a bunch of #include
    - needed to do some function/file refactoring to make it all work
- CUDA backend interoperable with pure CUDA
  - can call CUDA functions from Kokkos kernels
  - makes incremental porting and validation much easier
- All offloaded data structures need to be converted to Kokkos Views

Kokkos: Porting Data Structures

- Kokkos Views can either allocate host/device memory, or wrap existing pointers
  - makes incremental porting of cuMalloc memory easier
- Supports both row and column major ordering
- Jagged multidimensional arrays not well supported by Kokkos Views
  - Views of Views not meant for this
  - lots of extra boilerplate needed to make work
  - easier to flatten to 1D array, or pad to 2D
- Requires explicit Host ↔ Device memory migration
  - need to create Views on host to hold copied information
- Non-zero overhead to using Views
  - both in the extra steps for creating the host/device Views, and operations on them

Kokkos: Porting Kernel Code

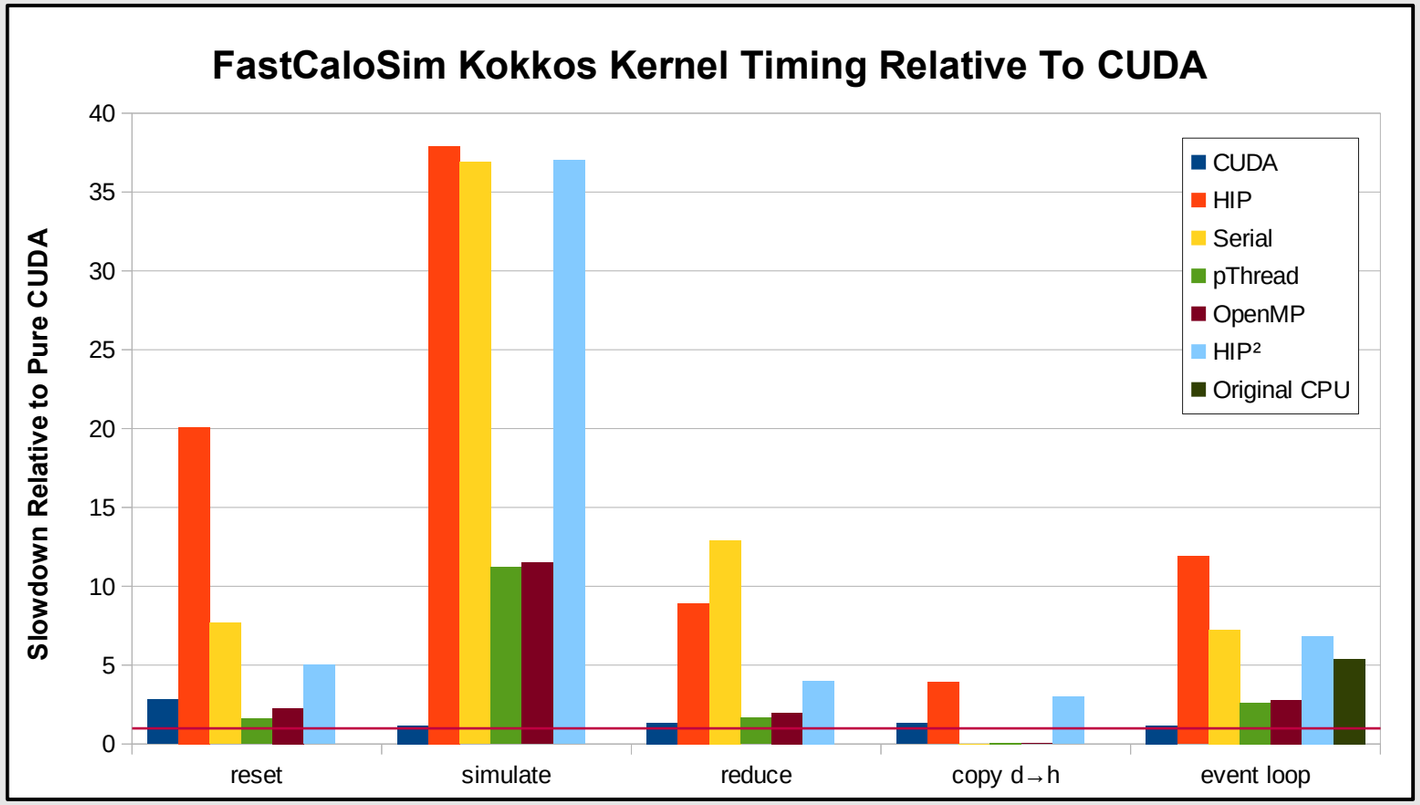
- While syntax is different from CUDA, concepts are the same
  - functions → lambdas
  - parallel_for, parallel_scan, reductions
    - some CUDA features not available in Kokkos (yet?). See Patatrack
  - atomics (but not between devices or host/device parallel execution spaces)

- Most FCS functions identical between CUDA / Kokkos
  - use a single file with #ifdef to select attributes to share as much code between version

```
#ifdef USE_KOKKOS
#  include <Kokkos_Core.hpp>
#  include <Kokkos_Random.hpp>
#  define __DEVICE__ KOKKOS_INLINE_FUNCTION
#else
#  define __DEVICE__ __device__
#endif
```

Kokkos: Performance

- Exercise various backends, compare to original CUDA
  - CUDA reference is NVidia 2080
  - HIP on AMD GPU (Vega56)
  - Intel XeLP GPU via OpenMP target offload, but I can't show that…
  - pThread / OpenMP best performance with ~15 threads/procs
  - **HIP²** is a pure HIP port, run on AMD GPU



- Kokkos does not handle GPU memory initialization efficiently
- Kokkos kernel launch penalties worse than CUDA
- AMD Vega56 has horrible launch latencies
- HIP/AMD uses the CPU a lot more than CUDA when executing kernels on GPU
- Code was **ported** from CUDA, not rewritten
  - likely considerable optimization possible

CPU Freq	CUDA	Kokkos		HIP²
		CUDA	HIP	
2200 MHz	5.6	9.4	152	88
3700 MHz	3.4	5.3	60	30

kernel launch latencies / µs

SYCL Build Infrastructure

- Multiple different flavours of dpcpp/SYCL
  - Intel official “beta” releases
  - Intel closed codedrops at jlse for A21 development
    - OpenCL and LevelZero backends
  - Intel/llvm git
    - CUDA backend available (selectable at compile time)
    - RNG issues (See Vince’s talk later)
  - Codeplay
  - hipSYCL (AMD), triSYCL

- In theory SYCL is single source, compile once, run anywhere, select backend at runtime
  - in practice need to build with different compilers to target different hardware
  - maybe there will be convergence in the future
- Integrates well with CMake

SYCL Code Modifications

```
#ifdef SYCL_TARGET_CUDA
class CUDASelector : public cl::sycl::device_selector {
 public:
  int operator()(const cl::sycl::device& device) const override {
    const std::string device_vendor = device.get_info<device::vendor>();
    const std::string device_driver =
        device.get_info<cl::sycl::info::device::driver_version>();

    if (device.is_gpu() &&
        (device_vendor.find("NVIDIA") != std::string::npos) &&
        (device_driver.find("CUDA") != std::string::npos)) {
      return 1;
    };
    return -1;
  }
};
#endif
```

Custom selector: Select devices (targetable soon! :) based on driver information.

Context-sharing: When you create multiple `queue`s, even from the same `device`, a new `context` gets created each time. As such, any `buffer` (or allocated memory) created from a given context will be bound to that `context` (c.f. CUDA contexts).

```
  // Initialize device, queue and context
  cl::sycl::device dev;
  // Initialize device, queue and context
  if (!ctx_) {
    dev = fastcalosycl::syclcommon::GetTargetDevice();
    // dev = cl::sycl::device(cl::sycl::default_selector());
    queue_ = cl::sycl::queue(dev);
    ctx_ = new cl::sycl::context(queue_.get_context());
  } else {
    dev = ctx_->get_devices()[0];
    queue_ = cl::sycl::queue(*ctx_, dev);
  }
```

```
class SimResetKernel {
 public:
  SimResetKernel() = delete;
  SimResetKernel(syclcommon::SimProps* props)
      : num_cells_(props->num_cells), num_unique_hits_(nullptr) {
    SimHitRng* rng = (SimHitRng*)props->rng;
    num_unique_hits_ = rng->get_num_unique_hits();
    cells_energy_ = rng->get_cells_energy();
  }

  void operator()(cl::sycl::nd_item<1> item) const {
    unsigned int wid = item.get_global_linear_id();
    if (wid < num_cells_) {
      cells_energy_[wid] = 0.0;
    }
    if (wid == 0) {
      *num_unique_hits_ = 0;
    }
  }

 private:
  const unsigned long num_cells_;
  int* num_unique_hits_;
  float* cells_energy_;
};
```

Kernel function objects: Callables that are instantiated within a command group handler, and called directly via `single_task`, `parallel_for`, etc. for kernel invocation (c.f. writing lambda “inline”)

```
// Copyright (C) 2002-2020 CERN for the benefit of the ATLAS collaboration

// Storage of passive simulation data used during on-device simulation.
// These properties are set by different class objects before being transferred
// to the SYCL device for processing.

#ifndef FASTCALOSYCL_SYCLCOMMON_PROPS_H_
#define FASTCALOSYCL_SYCLCOMMON_PROPS_H_

#include <SyclCommon/Histo.h>

namespace fastcalosycl::syclcommon {

static const unsigned int kMinHits = 1000;
static const unsigned int kMaxHits = 200000;
static const unsigned int kMaxBins = 1024;
static const unsigned int kMaxUniqueHits = 2000;

struct CellProps {
  unsigned long cell_id;
  float energy;
};

struct SimProps {
  // Particle properties
  int pdgId;
  double charge;
```

Simplified data structures: In addition to virtual function calls, function pointers, exceptions, …, the SYCL 2020 spec. does not support non-standard-layout types.

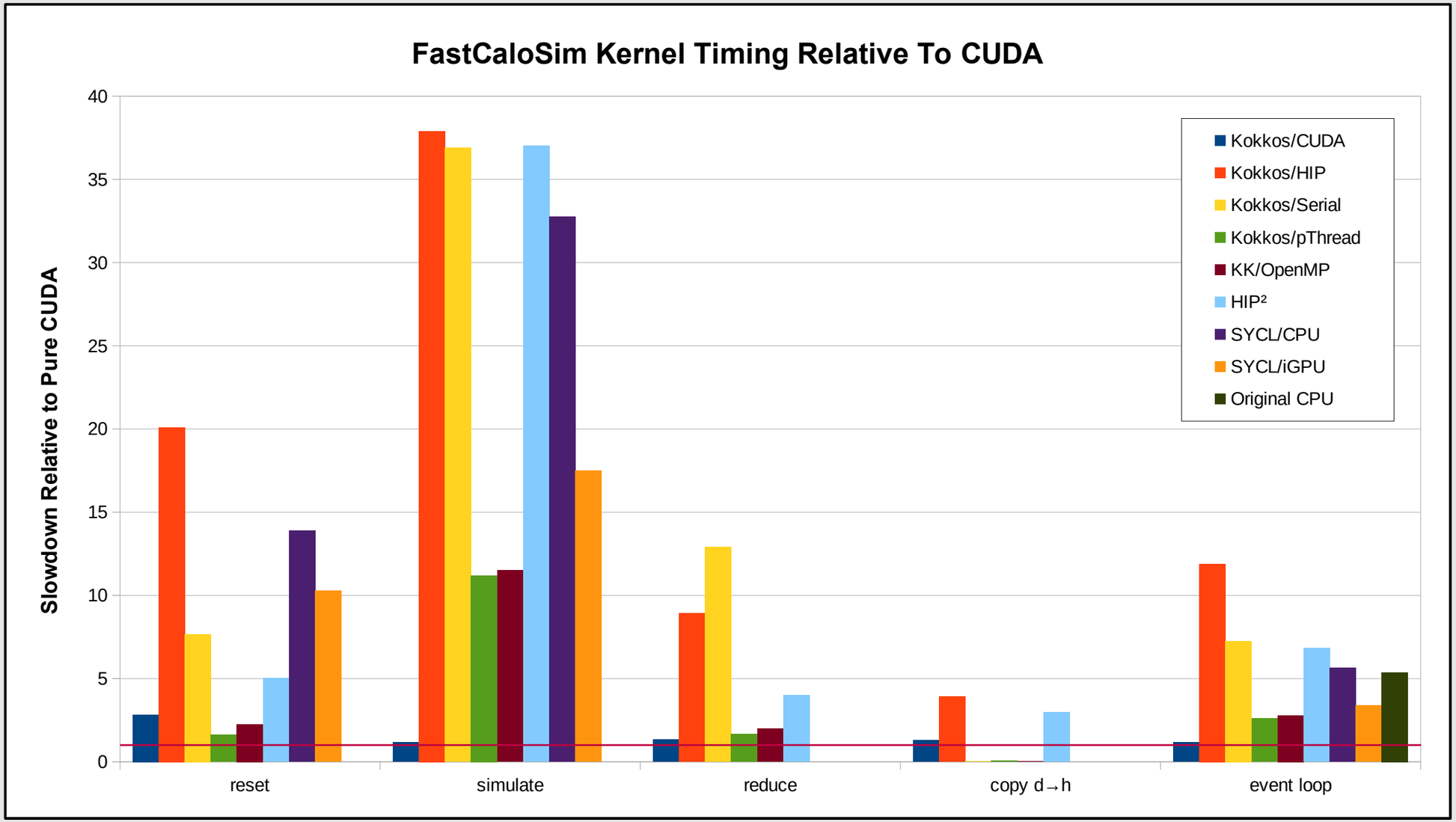
The ***DPC++ toolchain*** has ***undergone numerous improvements***, and ***useful extensions*** were added, over the past year -- this is reflected by the new ***429-page SYCL 2020 specification*** (c.f. the 274-page 1.2.1 specification). In particular, ***Unified Shared Memory*** (USM) was only a proposal about 6 months ago and is now part of the 2020 specificification, as well as support for ***floating-point type atomic operations***, leading to faster and easier development (not so many private builds of intel/llvm).

Unit tests: Validation of host and device geometries, ensure reproducible random numbers produced on different devices

- CMakeLists.txt
- GeoRegion_test.cxx
- Geo_test.cxx
- Histo_test.cxx
- SimEvent_test.cxx
- SimHitRng_test.cxx

SYCL Performance

- Timing tests on an integrated Intel GPU (Iris Pro P580) w/ public dpcpp beta10 release
- d→h transfer speeds are RAM→RAM so don't count



Lessons Learned

- Build configuration requirements may be challenging
  - Kokkos shared libs *vs* relocatable device code: code reorganization
  - dpcpp changing to (too?) rapidly, things that worked last week may not work today
- Separate binaries for different device backends
  - Kokkos explicitly, SYCL because you need different compiler flavours
  - implications for production code distribution
- CUDA→ Portability Layer concepts translate well
  - Views / Buffers come with overhead / penalties
- Launch latencies for tiny kernels kills performance on all platforms
  - Portability layers make it worse
  - AMD is really bad. Will RDNA2 / CDNA2 / Instinct improve things?
- High performance single source CPU/GPU may be a pipe dream
- GPU very underutilized in FastCaloSim
  - grouping data between events helps: may require significant refactoring of frameworks
  - a single GPU can be shared between multiple processes

What Recommendations Would I Make Today?

- Caveats:
  - non-NVidia/CUDA market very fast moving target

- Are you buying hardware for you trigger farm today?
  - NVidia / CUDA
- Is short term performance the main metric?
  - NVidia / CUDA
- Is short term performance important, but not ultimate, and want some portability?
  - Kokkos
- Do you want to target mainly Intel and NVidia GPU hardware?
  - SYCL
- Long term portability on all platforms
  - Kokkos

- **Non-NVidia software/hardware changing very rapidly: these answers may be different in six months.**

What Comes Next

- Other Parallel Portability Layers:
  - OpenMP / OpenACC
  - Alpaka
  - Raja - Will we learn anything that Kokkos didn't teach us?

- Other backends
  - SYCL w/ CUDA on NVidia
  - Intel discrete GPU (Arctic Sound/XeHP and Ponte Vecchio/XeHPC via Kokkos and SYCL)
    - we can already run FCS/SYCL on XeLP, XeHP nodes at jlse
  - AMD RDNA2 / CDNA2

- Better understanding/evaluation/reporting of metrics
  - in coordination with other testbeds

- Update FastCaloSim to reflect what ATLAS is currently using
  - more realistic particle scenarios
  - integrate into ATLAS repositories

Acknowledgments

- Really want to thank all the people who contributed to this project

- CUDA port:
  - Zhihua Dong (BNL)
  - Meifeng Lin (BNL)
  - Kwangmin Yu (BNL)
- Kokkos port:
  - Zhihua Dong (LBL)
  - Charles Leggett (LBL)
- SYCL port:
  - Charles Leggett (LBL)
  - Vincent Pascuzzi (LBL)
- Physics Validation:
  - Doug Benjamin (ANL)
  - Tadej Novak (DESY)

fin